Amendments to the Specification:

Please replace paragraph [0001] with the following amended paragraph:

[0001] This Application claims the benefit of U.S. Provisional Application No. 60/294,913 filed 5/31/2001, ~~the disclosure of~~ which is incorporated herein by reference.

Please replace paragraph [0003] with the following amended paragraph:

[0003] The present invention is related to the compilation of platform neutral bytecode computer instructions, such as JAVA, into high quality machine code. More specifically, the present invention discloses a new method of creating high quality machine code from platform neutral bytecode in a single sequential pass in which information from preceding instruction translations is used to mimic an optimizing ~~complier~~ compiler without the extensive memory and time requirements.

Please replace paragraph [0004] with the following amended paragraph:

[0004] The present invention is related to the compilation of platform neutral bytecode computer The benefit of architecture neutral language such as JAVA is the ability to execute such language on a wide range of systems once a suitable implementation technique, such as a JAVA Virtual Machine, is present. The key feature of the JAVA language is the creation and use of platform neutral bytecode instructions, which create the ability to run JAVA programs, such as applets, applications or servlets ~~servelets~~, on a broad range of diverse platforms. Typically, a JAVA program is compiled through the use of a JAVA Virtual Machine (JVM) which is merely an abstract computing machine used to compile the JAVA program (or source code) into platform neutral JAVA bytecode instructions, which are then placed into class files. The JAVA bytecode instructions in turn, serve as JVM instructions wherever the JVM is located. As bytecode instructions, the JAVA program may now be transferred to and executed by any system with a compatible JAVA platform. In addition, ~~any other language which may be~~ other languages that are expressed in bytecode instructions, may be ~~used~~ compatible with the JVM.

Please add a Summary section after the Background section:

SUMMARY

[0012.1] A technique for fast compilation of bytecode involves sequentially processing the bytecode in a single sequential pass in which information from preceding instruction translations is used to perform the same optimizing process of an optimizing compiler without the extensive memory and time requirements. The system for implementing the technique may include a development or target computer system including a computer readable storage medium with a compilation program included therein. One or more class files having one or more methods containing bytecode instruction listings may be compiled using the compilation program.

[0012.2] An example of such a compilation program includes multiple instruction sets. A first instruction set may, for example, be configured to select a first class to compile. A second instruction set may, for example, be configured to select a first method of the first class to compile. A third instruction set may, for example, be configured to select a first instruction to compile. Another instruction set may be for sequentially executing each bytecode instruction of the selected method.

[0012.3] As another example, a first instruction set may, for example, create map storage to store actual mappings and native code addresses. A second instruction set may initialize stack mappings to "empty" and addresses to "unknown". A third instruction set may sequentially select each bytecode instruction in each method of each class file. A fourth instruction set may detect stored stack mappings for the selected bytecode instruction.

[0012.4] As another example, a first instruction set may, for example, detect direct control flow from a bytecode instruction previous to a selected bytecode instruction. A second instruction set may store all stacks and set stack mappings to "stack" if direct control flow from the previous bytecode instruction is detected. A third instruction set may read a stack layout from the stack mappings and set the stack mappings to "stack" if direct control flow is not detected.

[0012.5] As another example, an instruction set may, for example, set a native code address for a bytecode instruction.

[0012.6] As another example, a first instruction set may, for example, detect if an actual instruction is a load constant instruction. A second instruction set may create a new constant stack mapping if the actual instruction is a load constant instruction.

[0012.7] As another example, a first instruction set may, for example, detect if an actual instruction is a load local instruction. A second instruction set may create a new local stack mapping if the actual instruction is a load local instruction.

[0012.8] As another example, a first instruction set may, for example, detect if an actual instruction is a stack manipulating instruction. A second instruction set may duplicate or reorder a stack mapping according to the stack manipulating instruction.

[0012.9] As another example, a first instruction set may, for example, detect if an actual instruction is a jump or switch instruction. A second instruction set may emit code using stack mapping information if the actual instruction is a jump or switch instruction. A third instruction set may store unused stack values.

[0012.10] As another example, a first instruction set may, for example, detect if an actual instruction is some other instruction. A second instruction set may emit code using stack mapping information if the actual instruction is some other instruction.

[0012.11] As another example, a first instruction set may, for example, select a next instruction. A second instruction set may select a next method. A third instruction set may select a next class file.

[0012.12] A method implementing the technique may, for example, include receiving a bytecode listing onto a computer readable medium containing compilation procedure instructions, executing the compilation procedure instructions to sequentially processing each bytecode of the bytecode listing, and produce native machine code on the computer readable medium, using preceding translation information to optimize the native machine code.

Please replace paragraph [0013] with the following amended paragraphs:

[0013] These and other objects, features and characteristics of the present invention will become more apparent to those skilled in the art from a study of the following detailed description in conjunction with the appended claims and drawings, all of which form a part of this

specification. The embodiments and figures are illustrative rather than limiting; they provide examples of the invention. The invention is limited only by the claims.

~~FIG. 1A (prior art) illustrates a flowchart of traditional bytecode instruction first pass compilation;~~

~~FIG. 1B (prior art) illustrates a flowchart of traditional bytecode instruction second pass compilation;~~

~~FIG. 2 illustrates a flowchart of the embodiment of the new method; and~~

~~FIG. 3 illustrates the data structures required by the new method.~~

[0013.1] FIG. 1A (Prior Art) depicts a flowchart of traditional bytecode instruction first pass compilation.

[0013.2] FIG. 1B (Prior Art) depicts a flowchart of traditional bytecode instruction second pass compilation.

[0013.3] FIGS. 2A and 2B depict a flowchart of an embodiment of the new method.

[0013.4] FIG. 3 depicts a conceptual view of data structures for use with the new method.

Please replace paragraph [0016] with the following amended paragraph:

[0016] In prior art ~~Figures~~ FIGS. 1A and 1B, an illustrative flow diagram of a simple bytecode translator compilation method is shown. In prior art ~~Figure~~ FIG. 1A, a traditional compilation method is shown as flow diagram 100 which loops through the bytecode instructions, analyzing an individual bytecode instruction during each loop as stated in step 102. After each bytecode instruction is analyzed, the method determines the stack status from the bytecode instruction being analyzed and stores the stack status in stack status storage as stated in step 104. When the last bytecode instruction is analyzed as stated step 102, the loop is ended at step 108 and partial compilation is completed.

Please replace paragraph [0017] with the following amended paragraph:

[0017] In prior art ~~Figure~~ FIG. 1B, remaining compilation occurs in flow diagram 150 which shows farther loops through the bytecode instructions analyzing an individual bytecode

instruction during each loop as stated in step 152. The stack status storage and bytecode instruction are then used to translate the bytecode instruction into machine code as stated in step 154. When the last bytecode instruction is translated as stated in step 152, the loop is ended at step 158 and compilation is completed.

Please replace paragraph [0018] with the following amended paragraph:

[0018] ~~In Figure 2,~~ FIGS. 2A and 2B depict an illustrative flow diagram 200 of an embodiment of the new method ~~of the new method is shown in flow diagram 200~~. In step 202, a class file placed on the development or target system is selected ~~and~~. In step 204, a first method within the first class file is selected ~~in step 204. At this point, storage and initialization occurs in step 206~~. In step 205, storage is allocated to facilitate storage of actual stack mappings and native code addresses. In step 206, stack mappings are initialized to "empty" and addresses are initialized to "unknown". In step ~~208~~ 207, a first bytecode instruction is selected and evaluated ~~to determine if~~. At decision block 208, it is determined whether there is a stack ~~map~~ mapping stored for the actual bytecode instruction. If ~~and, if~~ there is a stored stack ~~map~~ mapping (208-Y), then at decision block 210 it is determined whether ~~, step 210 determines if there is~~ direct control flow from the previous instruction is present. ~~If there is no direct control flow present, step 214 results in reading the stack layout from the stack map in bytecode and setting mappings to 'stack'.~~ If direct control flow is present (210-Y) or if there is not a stored stack mapping (208-N), then at step 212 code is emitted ~~, step 212 produces a code~~ to store all stacks and set their stack mappings to 'stack'. If direct control flow is not present (210-N), then at step 214 stack layout is read from the stack mapping and mapping is set to stack. After either step 212 or step 214, in ~~In~~ step 216, the native code address for the actual instruction is ~~then~~ set.

Please replace paragraph [0019] with the following amended paragraph:

[0019] In an embodiment, the flowchart 200 continues at decision block 218 where it is determined whether the instruction is "load constant". If the instruction is "load constant" (218-Y), then at step 220 a new constant stack mapping is created and the flowchart 200 continues from decision block 238, which is described later. If the instruction is not "load constant" (218-N), then at decision block 222 it is determined whether the instruction is "load local". If the instruction is "load local" (222-Y), then at step 224 a new local stack mapping is created and the flowchart 200 continues from decision block 238. If the instruction is not "load local" (222-N),

then at decision block 226 it is determined whether the instruction is a stack manipulating instruction. If the instruction is a stack manipulating instruction (226-Y), then at step 228 the stack mapping is duplicated and/or reordered according to the instruction and the flowchart 200 continues from decision block 238. If the instruction is not a stack manipulating instruction (226-N), then at step 230 it is determined whether the instruction is a jump or switch instruction. If the instruction is a jump or switch instruction (230-Y), then at step 232 a code is emitted using the stack mapping information, at step 234 a code is emitted to store unused stack values, and the flowchart 200 continues from decision block 238. It should be noted that unused stack values, as used herein, are the values represented in the various stacks that have not yet been translated into codes. If the instruction is not a jump or switch instruction (230-N), then at step 236 a code is emitted using stack mapping information to locate the arguments and the flowchart 200 continues at decision block 238. The mappings for the arguments are removed and new mappings are created if the instruction results in a new value. ~~In steps 218 and 222, the instruction is evaluated to determine if the actual instruction is 'load constant' or load local'. If the instruction is load constant, step 220 creates a new constant stack mapping. If the instruction is load local, step 224 creates a new local stack mapping. If the instruction is a stack manipulating instruction as determined in step 226, stack mappings are duplicated or reordered in step 228 according to the instruction. If the actual instruction is a jump or switch instruction as determined in step 230, a code is produced for the actual instruction using stack mapping information and a code is produced to store all stack values not used in step 232.~~

Please replace paragraph [0020] with the following amended paragraph:

[0020] In an embodiment, the flowchart 200 continues at decision block 238 where it is determined whether a next instruction is available. If a next instruction is available (238-Y), then at step 244 the next instruction is selected and the flowchart 200 continues from decision point 208, described previously. If a next instruction is not available (238-N), for example because the instruction is a last instruction of a method or a class, then the flowchart 200 continues at decision block 240 where it is determined whether a next method is available. If a next method is available (240-Y), then at step 246 the next method is selected and the flowchart 200 continues from step 205, described previously. If a next method is not available (240-N), for example because the method was the last method of a class, then the flowchart 200 continues at decision block 242 where it is determined whether a next class is available. If a next class is

available (242-Y), then at step 248 the next class is selected and the flowchart 200 continues from step 204, described previously. If a next class is not available (242-N), then it is assumed in this embodiment that each instruction of each method of each class has been processed and the flowchart 200 ends. ~~In step 234, if the actual instruction is any other instruction , a code is produced for the actual instruction using stack mapping information to locate the arguments in step 236. The mappings for the arguments are removed and new mappings are created if the instruction results in a new value.~~

Please replace paragraph [0021] with the following amended paragraph:

[0021] It should be noted that decision blocks need not indicate a decision is made at each block. For example, a determination of whether the instruction is a "load constant" (see, e.g., decision block 218 of FIG. 2B) or a "load local" (see, e.g., decision block 222 of FIG. 2B) could be made as a single decision, rather than two separate decisions. This method and other methods are depicted as serially arranged modules. However, modules of the methods may be reordered, combined, or arranged for parallel execution as appropriate. ~~Once the instruction has been analyzed, the following bytecode instruction is selected for translation. If there are no remaining instructions, the next method is selected in step 204 and if there are no remaining methods, the next class is selected in stop 202. If there are no remaining classes, the evaluation returns in step 238.~~

Please replace paragraph [0023] with the following amended paragraph:

[0023] ~~Referring now to Table 1, the new fast compilation method places each class file in the development or target system, at which point each method in the class containing bytecode instructions is analyzed~~ Referring now to FIG. 3, an embodiment of the new fast compilation method may be executed on a development or target system that includes one or more class files. A class file contains one or more methods with bytecode instructions. According to an embodiment, the bytecode instructions of each method are analyzed. Storage and data structures for actual mappings and native code addresses are created and the stack mappings are initialized to ~~empty~~ "empty" and addresses are initialized to ~~unknown~~ "unknown". Each bytecode instruction, from first to last is then evaluated and translated into high quality machine code.

Please replace paragraph [0025] with the following amended paragraph:

[0025] The sequential bytecode instructions are then evaluated to determine if the actual instruction is 'load constant', ~~load local'~~ 'load local', a stack manipulating instruction, a jump, switch or any other instruction. If the actual instruction is load constant, a new constant stack mapping is created. If however, the actual instruction is load local, new local stack mapping is created.

Please replace paragraph [0028] with the following amended paragraph:

[0028] Prior art methods such as simple translators and optimizing compilers fail to produce the results associated with the new method. Through the use of a sequential pass, the simplicity and speed of simple translators is achieved. As used herein, the language "mimic an optimizing compiler" refers to the utilization of information from the translation of preceding bytecodes to compile a sequence of bytecodes into machine code in a sequential pass, an example of which is described above with reference to FIGS. 2A and 2B. As used herein, the language "using information from preceding instructions" refers to using information that is stored in stacks as described, for example, with reference to FIGS. 2A and 2B, when processing a current bytecode. Data structures appropriate for utilization in this regard are conceptually depicted, for example, in FIG. 3. However the use of preceding translation information to mimic optimizing compilers when possible, creates the high quality machine code translation ~~of the~~ characteristic of an optimizing compiler. Therefore the present invention produces a high quality translation with greater simplicity and speed than previously known.